DT505G: Algorithms, Data Structures and Complexity

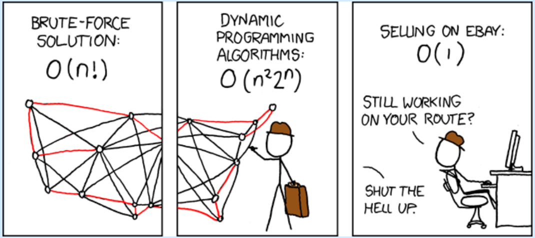
Introduction

Federico Pecora

federico.pecora@oru.se

Center for Applied Autonomous Sensor Systems (AASS)

Örebro University, Sweden



© XKCD

Staff and Practicalities

- Staff
  - **Lectures, course responsible:** Federico Pecora
    room: T2215
    phone: 019303319
    email: `federico.pecora@oru.se`

  - **Lab assistant:** Uwe Köckemann
    room: T2237
    phone: 019303415
    email: `uwe.kockemann@oru.se`

- 12 lectures, 7 lab sessions (4 labs + 1 project)
  - addressing specific data structures
  - used to implement specific algorithms
  - all lab work realized in C

Teaching method

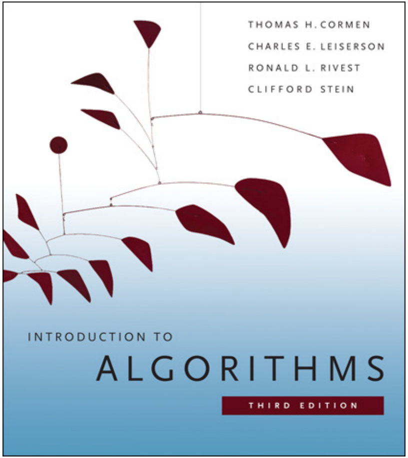
- Lectures will be whiteboard-based ⇝ *attending and taking notes is important!*

Teaching method

- Lectures will be whiteboard-based ⇝ *attending and taking notes is important!*

- Lectures closely follow course book
- Lectures are closely paired with labs ⇝ *"the devil is in the detail"*

Course Book



Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein

Introduction to Algorithms (3rd edition)

MIT Press, July 2009

ISBN:
9780262033848 (hardcover)
9780262533058 (paperback)
9780262259460 (eBook)

Labs and Project

- **Lab 1, 2:** Elementary Data Structures
- **Lab 3:** Sorting
- **Lab 4:** Trees and Recursion
- **Lab 5:** Working with Graphs / Dynamic Programming
- **Lab 6, 7:** Project work

Examination and Grading (Theory, 4.5 hp)

- Written exam, each exercise gives points
- Theoretical questions about data structures, algorithms, and complexity
- Exercises that involve formulating and/or solving problems
- Exercises that involve understanding an existing algorithm
- Grades: U, 3, 4, 5

Examination and Grading (Labs + Project, 1.5 hp + 1.5 hp)

- Exercises including implementation and theoretical questions
- All exercises to be demonstrated to instructors during lab sessions
- All code to be handed in to instructors
- Labs and project can be done in pairs
- Grades: U, G

What is an Algorithm?

Muḥammad ibn Mūsā al-Khwārizmī

محمد بن موسى خوارزمی

noun *algorithm*

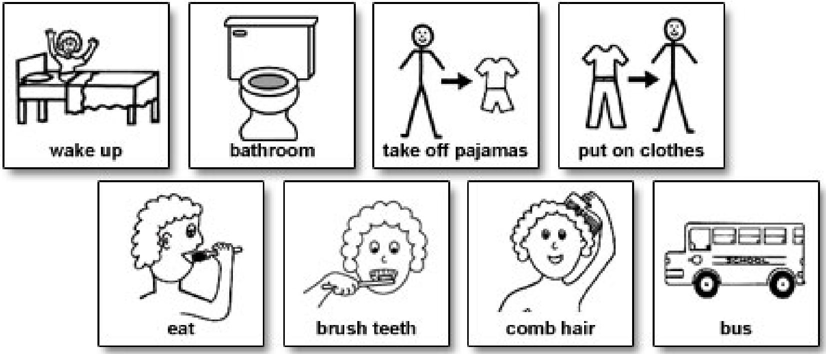
- 1690s, French *algorithme* "Arabic system of computation"
  - 13th century, Old French *algorisme* "the Arabic numeral system"
    - Medieval times, Latin *algorismus*
      - 780 AD, Arabic *al-Khwarizmi* (native of Khwarazm, Uzbekistan)

- *"Algoritmi de numero Indorum"* ("al-Khwārizmī on the Hindu Art of Reckoning"), ca. 820 AD
- Spreads the Hindu-Arabic numeral system
- Methods for solving linear and quadratic equations, fundamentals of Algebra, Arithmetic, Astronomy, Trigonometry, Geography

Algorithms in Daily Use

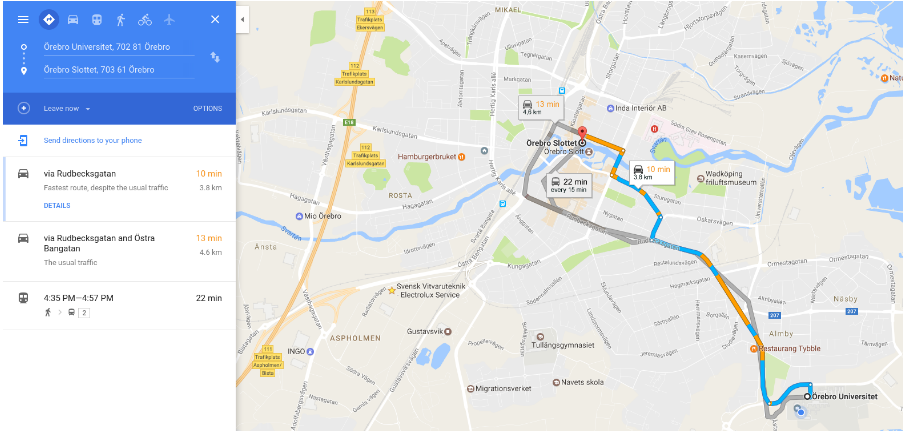
Google: a *process or set of rules* to be followed in calculations or other problem-solving operations, *especially* by a computer.

An Morning Algorithm

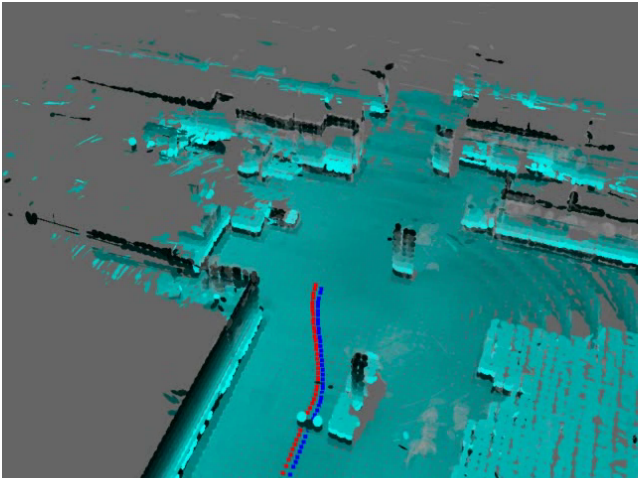


An Algorithm for Making Sandwitches

An Algorithm for Route Planning

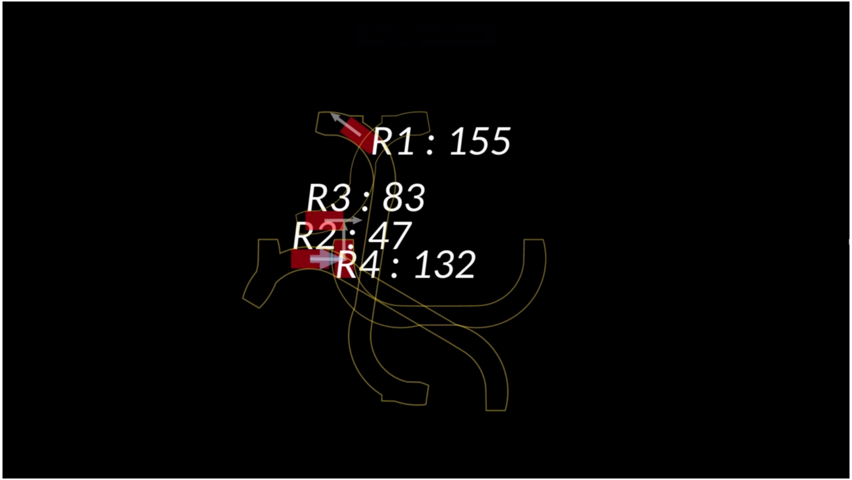


An Algorithm for Robot Navigation



An Algorithm for Robot Manipulation

An Algorithm for Multi-Robot Coordination



Euclid's Algorithm for Finding the Greatest Common Divisor

Input: Two positive integers, a and b.
Output: The greatest common divisor, g, of a and b.

1. If $a < b$, exchange a and b.
2. Divide a by b and get the remainder, r.
3. If $r = 0$, report b as the GCD of a and b.
4. Replace a by b and replace b by r.
5. Return to step 2.

What is an Algorithm?

Google: a *process or set of rules* to be followed in calculations or other problem-solving operations, *especially* by a computer.

Merriam-Webster: a procedure for solving a *mathematical problem* in a *finite number of steps* that frequently involves *repetition* of an operation.

Well-Known Algorithms for Well-Known Problems

- **Scenario:** a truck needs to visti n locations
- **Traveling salesperson problem:** find optimal tour visiting each location once

Well-Known Algorithms for Well-Known Problems

- **Scenario:** a truck needs to visti *n* locations
- **Traveling salesperson problem:** find optimal tour visiting each location once
- Underlies many different applications (from logistics to computer graphics)
- Can be seen as a **graph problem**

What is an Algorithm?

Google: a *process or set of rules* to be followed in calculations or other problem-solving operations, *especially* by a computer.

Merriam-Webster: a procedure for solving a *mathematical problem* in a *finite number of steps* that frequently involves *repetition* of an operation.

Cormen et al.: any well-defined *computational procedure* that takes some value (or set of values) as *input* and produces some value (or set of values) as *output*.

What Makes a Good Algorithm?

What Makes a Good Algorithm?

- **Correctness:** does the algorithm compute the correct result?

What Makes a Good Algorithm?

- **Correctness:** does the algorithm compute the correct result?
- **Efficiency:** how fast is the algorithm?

What Makes a Good Algorithm?

- **Correctness:** does the algorithm compute the correct result?

VS.

- **Efficiency:** how fast is the algorithm?

Good Algorithms and Bad Algorithms

- **Traveling salesperson problem:** find optimal tour visiting each location once

Good Algorithms and Bad Algorithms

- **Traveling salesperson problem:** find optimal tour visiting each location once
- n locations $\Rightarrow (n-1)!$ possible tours

Good Algorithms and Bad Algorithms

- **Traveling salesperson problem:** find optimal tour visiting each location once
- n locations $\Rightarrow (n-1)!$ possible tours
- 25 locations $\Rightarrow$ more than 6.2×10^{23} possible tours

Good Algorithms and Bad Algorithms

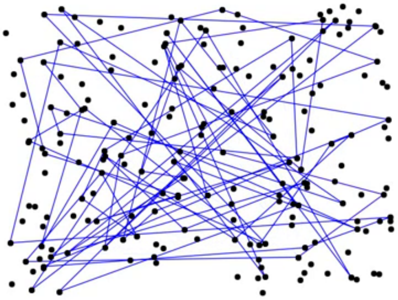
- **Enumeration:** 1 ms per tour $\Rightarrow$ 1.966×10^{16} years of computation

Good Algorithms and Bad Algorithms

- **Enumeration:** 1 ms per tour $\Rightarrow$ 1.966×10^{16} years of computation
- **Nearest insertion algorithm:** finds sub-optimal solution in seconds (time $\propto n^2$)

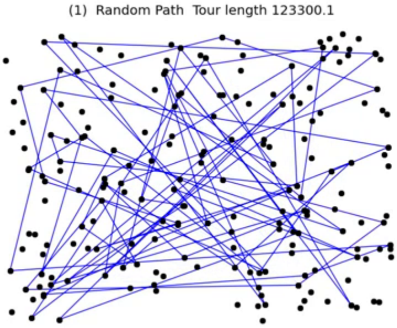
How Can We Measure Algorithm Performance?

(1) Random Path Tour length 123300.1



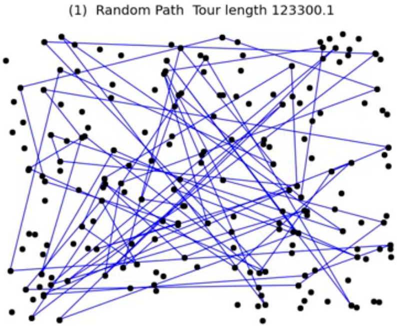
- Time to compute solution?

How Can We Measure Algorithm Performance?



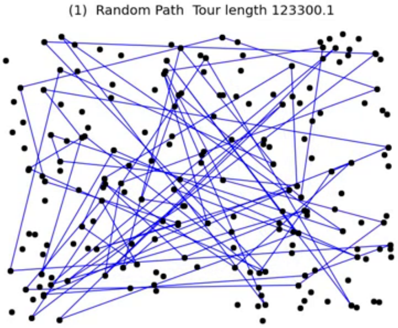
- Time to compute solution?
- But that only tells us

How Can We Measure Algorithm Performance?



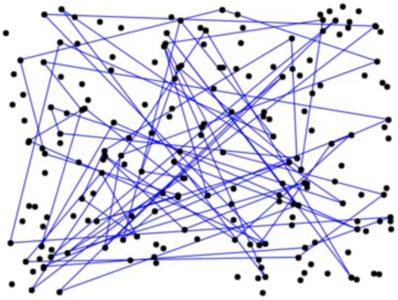
- Time to compute solution?
- But that only tells us
  - time to run a particular implementation

How Can We Measure Algorithm Performance?



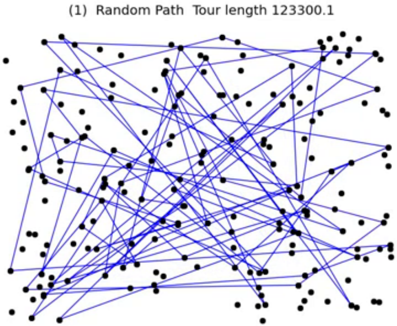
- Time to compute solution?
- But that only tells us
  - time to run a particular implementation
  - in a particular programming language

How Can We Measure Algorithm Performance?



- Time to compute solution?
- But that only tells us
  - time to run a particular implementation
  - in a particular programming language
  - on a particular computer

How Can We Measure Algorithm Performance?



- Time to compute solution?
- But that only tells us
  - time to run a particular implementation
  - in a particular programming language
  - on a particular computer
  - and just for the given input

Asymptotic Analysis

# nodes	Enumeration	Nearest insertion
1	1	1
2	1	4
3	2	9
4	6	16
...		
25	620448401733239439360000	625
...		
n	$n!$	n^2

How to Describe Algorithms? Code vs. Pseudocode

```
#include <stdio.h>

int main () {
  int array[] = {1, 7, 3, 4, 5};
  printf("Maximum␣is:␣%d\n", maximum(array,5));
  return 0;
}

/* Assumes all numbers in sequence >= 0 */
int maximum(int* sequence, int size) {
  int i, max = -1;
  for(i = 0; i < size; i++) {
    if (sequence[i] > max) {
      max = sequence[i];
    }
  }
  return max;
}
```

How to Describe Algorithms? Code vs. Pseudocode

```
MAXIMUM(A)
// Scan all of A
for i = 1 to A.length
    if A[i] > max
        max = A[i]
return max
```

How to Describe Algorithms? Code vs. Pseudocode

Input: a sequence A of n positive numbers $\langle a_1, a_2, \ldots, a_n \rangle$
Output: the maximum element a in sequence A

```
MAXIMUM(A)
// Scan all of A
for i = 1 to A.length
    if A[i] > max
        max = A[i]
return max
```

What is a Data Structure?

- **Data structure:** a way of collecting and organising data so that we can perform operations on these data in an effective way

What is a Data Structure?

- **Data structure:** a way of collecting and organising data so that we can perform operations on these data in an effective way
- Primitive data structures (aka, datatypes):
  - Integers, floats, booleans, characters, etc.

What is a Data Structure?

- **Data structure:** a way of collecting and organising data so that we can perform operations on these data in an effective way
- Primitive data structures (aka, datatypes):
  - Integers, floats, booleans, characters, etc.
- **Abstract data structures**:
  - Linked lists
  - Stacks
  - Queues
  - Trees
  - Graphs
  - ...

What is a Data Structure?

- **Data structure:** a way of collecting and organising data so that we can perform operations on these data in an effective way
- Primitive data structures (aka, datatypes):
  - Integers, floats, booleans, characters, etc.
- **Abstract data structures**:
  - Linked lists
  - Stacks
  - Queues
  - Trees
  - Graphs
  - ...

Dynamic sets:

- SEARCH(S,k)
- INSERT(S,x)
- DELETE(S,x)
- MINIMUM(S)
- MAXIMUM(S)
- SUCCESSOR(S,x)
- PREDECESSOR(S,x)

Algorithm Analysis: Complexity

Input: a sequence A of n positive numbers $\langle a_1, a_2, \ldots, a_n \rangle$
Output: the maximum element a in sequence A

MAXIMUM(A)

// Scan all of A
for $i = 1$ **to** $A.length$
    if $A[i] > max$
        $max = A[i]$
return max

Algorithm Analysis: Complexity

Input: a sequence A of n positive numbers $\langle a_1, a_2, \ldots, a_n \rangle$
Output: the maximum element a in sequence A

```
MAXIMUM(A)
1  // Scan all of A
2  for i = 1 to A.length
3      if A[i] > max
4          max = A[i]
5  return max
```

line	*cost*	*times*
1	c_1	1
2	c_2	$n+1$
3	c_3	n
4	c_4	t
5	c_5	1

$t =$ number of times test in line 3 succeeds

Model of Computation: RAM

- To analyze an algorithm we must assume a **computational model**

Model of Computation: RAM

- To analyze an algorithm we must assume a **computational model**
- **Random Access Machines**

Model of Computation: RAM

- To analyze an algorithm we must assume a **computational model**
- **Random Access Machines**
  - a simple register-based machine with indirect addressing

Model of Computation: RAM

- To analyze an algorithm we must assume a **computational model**
- **Random Access Machines**
  - a simple register-based machine with indirect addressing
  - instructions are executed sequentially

Model of Computation: RAM

- To analyze an algorithm we must assume a **computational model**
- **Random Access Machines**
  - a simple register-based machine with indirect addressing
  - instructions are executed sequentially
  - instruction set is inspired by real computers: arithmetic (addition, multiplication, . . .), data movement (load, store, copy), control (loops, conditionals, subroutine calls, return)

Model of Computation: RAM

- To analyze an algorithm we must assume a **computational model**
- **Random Access Machines**
  - a simple register-based machine with indirect addressing
  - instructions are executed sequentially
  - instruction set is inspired by real computers: arithmetic (addition, multiplication, ...), data movement (load, store, copy), control (loops, conditionals, subroutine calls, return)
  - data types are integers, floats ($c \ln n$ bits, $c \geq 1$, c finite)

Model of Computation: RAM

- To analyze an algorithm we must assume a **computational model**
- **Random Access Machines**
  - a simple register-based machine with indirect addressing
  - instructions are executed sequentially
  - instruction set is inspired by real computers: arithmetic (addition, multiplication, ...), data movement (load, store, copy), control (loops, conditionals, subroutine calls, return)
  - data types are integers, floats ($c \ln n$ bits, $c \geq 1$, c finite)
  - instructions have **unit cost**

Model of Computation: RAM

- To analyze an algorithm we must assume a **computational model**
- **Random Access Machines**
  - a simple register-based machine with indirect addressing
  - instructions are executed sequentially
  - instruction set is inspired by real computers: arithmetic (addition, multiplication, ...), data movement (load, store, copy), control (loops, conditionals, subroutine calls, return)
  - data types are integers, floats ($c \ln n$ bits, $c \geq 1$, c finite)
  - instructions have **unit cost**
- Hence, running time = number of operations executed

Algorithm Analysis: Complexity

Input: a sequence A of n positive numbers $\langle a_1, a_2, \ldots, a_n \rangle$
Output: the maximum element a in sequence A

```
MAXIMUM(A)
1  // Scan all of A
2  for i = 1 to A.length
3        if A[i] > max
4              max = A[i]
5  return max
```

line	*cost*	*times*
1	0	1
2	1	$n+1$
3	1	n
4	1	t
5	1	1

$$T(n) = (n+1) + (n) + (t) + (1)$$

Worst case: $t = n \Rightarrow T(n) = 3n + 2$

Best case: $t = 1 \Rightarrow T(n) = 2n + 3$

Algorithm Analysis: Complexity

Input: a sequence A of n positive numbers $\langle a_1, a_2, \ldots, a_n \rangle$

Output: the maximum element a in sequence A

```
MAXIMUM(A)
1  // Scan all of A
2  for i = 1 to A.length
3      if A[i] > max
4          max = A[i]
5  return max
```

line	*cost*	*times*
1	0	1
2	1	$n+1$
3	1	n
4	1	t
5	1	1

$$T(n) = (n+1) + (n) + (t) + (1)$$

Worst case: $t = n \Rightarrow T(n) = 3n + c$

Best case: $t = 1 \Rightarrow T(n) = 2n + c$

Algorithm Analysis: Complexity

Input: a sequence A of n positive numbers $\langle a_1, a_2, \ldots, a_n \rangle$
Output: the maximum element a in sequence A

MAXIMUM(A)

```
1  // Scan all of A
2  for i = 1 to A.length
3      if A[i] > max
4          max = A[i]
5  return max
```

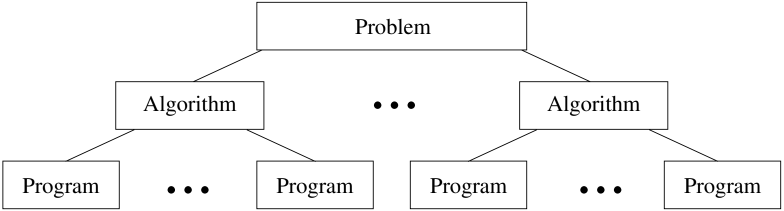
line	*cost*	*times*
1	0	1
2	1	$n+1$
3	1	n
4	1	t
5	1	1

$$T(n) = (n+1) + (n) + (t) + (1)$$

Worst case: $t = n \Rightarrow T(n)$ is linear

Best case: $t = 1 \Rightarrow T(n)$ is linear

Algorithm Complexity



Algorithm Complexity

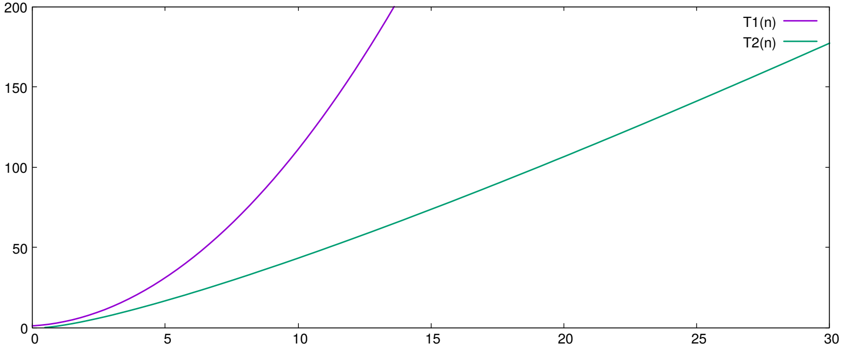
- Algorithm complexity: "at what rate does work grow with size n of input?"
  - Constant: $T(n) = c$
  - Sub-linear: $T(n) = c_1 \log n + c_2$
  - Linear: $T(n) = c_1 n + c_2$
  - Nearly linear: $T(n) = c_1 n c_2 \log n + c_3$
  - Quadratic: $T(n) = c_1 n^2 + c_2 n + c_3$
  - Exponential: $T(n) = c_1 x^n + c_2 x^{n-1} + \cdots + c_n$
  - ...

Algorithm Complexity

- Algorithm complexity: “at what rate does work grow with size n of input?”
  - Constant: $T(n) = c$
  - Sub-linear: $T(n) = c_1 \log n + c_2$
  - Linear: $T(n) = c_1 n + c_2$
  - Nearly linear: $T(n) = c_1 n c_2 \log n + c_3$
  - Quadratic: $T(n) = c_1 n^2 + c_2 n + c_3$
  - Exponential: $T(n) = c_1 x^n + c_2 x^{n-1} + \cdots + c_n$
  - ...
- But, for a given problem, how do we know **if a better algorithm is possible**?
  - such questions are studied in the field of **problem complexity**
  - this is outside the scope of this course

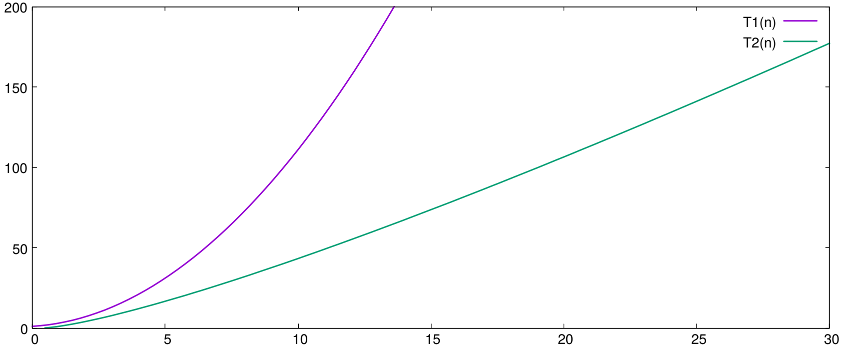
Algorithm Complexity

- Insersion sort: $T_1(n) = c_1 n^2 + c_2 n + c_3$ (quadratic)
- Merge sort: $T_2(n) = c_1 n + c_2 n \log_2 n$ (nearly linear)



Algorithm Complexity

- Insersion sort: $T_1(n) = c_1 n^2 + c_2 n + c_3$ (quadratic)
- Merge sort: $T_2(n) = c_1 n + c_2 n \log_2 n$ (nearly linear)



- Can we do **better than nearly linear**?

Algorithm and Problem Complexity

- **Algorithmic complexity** is defined by **analysis of an algorithm** (asymptotic analysis)
- **Problem complexity** is defined by
  - An upper bound, defined by an **algorithm**
  - A lower bound, defined by a **proof**
- **Remember:** algorithm and problem complexity are two **different**, but related, concepts!

Overview of Course Contents

- Introduction (algorithms vs. their implementation; why study algorithms and their complexity?)
- Elementary data structures and algorithms (lists, stacks, queues, sorting)
- Complexity of algorithms (asymptotic analysis)
- Advanced data structures (binary trees)
- Recursion (e.g., visiting trees)
- Complexity of problems (Turing machines, complexity classes)
- Algorithm design (divide and conquer, dynamic programming, graph search)

Thank you!

?